UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/551,454	10/01/2008	Roy Oberhauser	32860-000953/US	3037

30596 7590 02/17/2011

HARNESS, DICKEY & PIERCE, P.L.C.
P.O.BOX 8910
RESTON, VA 20195

EXAMINER	
SWIFT, CHARLES M	

ART UNIT	PAPER NUMBER
2196	

NOTIFICATION DATE	DELIVERY MODE
02/17/2011	ELECTRONIC

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

Notice of the Office communication was sent electronically on above-indicated "Notification Date" to the following e-mail address(es):

dcmailroom@hdp.com
siemensgroup@hdp.com
pshaddin@hdp.com

Office Action Summary	Application No.	Applicant(s)	
	10/551,454	OBERHAUSER ET AL.	
	Examiner	Art Unit	
	Charles Swift	2196	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE *3* MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on *18 January 2011*.

2a)☒ This action is **FINAL**. 2b)☐ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1 - 29* is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) *1 - 29* is/are rejected.

7)☐ Claim(s) _____ is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☐ The specification is objected to by the Examiner.

10)☐ The drawing(s) filed on _____ is/are: a)☐ accepted or b)☐ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☐ All b)☐ Some * c)☐ None of:

1.☐ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____.

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

1) ☐ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☐ Information Disclosure Statement(s) (PTO/SB/08) Paper No(s)/Mail Date _____.
4) ☐ Interview Summary (PTO-413) Paper No(s)/Mail Date. _____ .
5) ☐ Notice of Informal Patent Application
6) ☐ Other: _____.

DETAILED ACTION

1. This Office Action is based on the application filed on 1/18/2011.

2. Claims 1 - 29 are pending.

3. Claims 1, 6, 11, 13, 18, 22 – 27 and 29 are amended.

4. Objections to claims 1, 11, 25, 26, 27 and 29 are withdrawn in view of applicant's amendment.

5. Claim rejection under 35 USC 112 applied to claim 24 is withdrawn in view of applicant's amendment.

6. Claim rejection under 35 USC 101 applied to claims 22 – 25 is withdrawn in view of applicant's amendment.

Claim Rejections - 35 USC § 102

7. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (b) the invention was patented or described in a printed publication in this or a foreign country or in public use or on sale in this country, more than one year prior to the date of application for patent in the United States.

8. Claims **1 – 4, 22, 26 and 27** are rejected under 35 U.S.C. 102(b) as being anticipated by **Waldin et al (USPAT 6052531, hereinafter Waldin)**.

As per claim 1, Waldin discloses: A method for modifying software, comprising:

- Initially forming, from an original piece of software including only source text, a hybrid form of the original software, formed in such a way that at least one part of

the source text is compiled into at least one of a byte and binary code and at least one further part of the source text is converted into a code formulated in a meta markup language for at least one variation point. (Waldin col 4, line 17 – 35, "Each time an updated software application 110 is produced by the virus protection software publisher, the updated form of the software application constitutes a new version. The software publisher uses an incremental update builder, such as binary patch file builder 120, to produce at least one incremental update, such as binary patch file 122, which can transform a previous version of the software application to the current version. A binary patch file builder 120 is a program which takes two versions of a software application, for example versions A and B, and produces a binary patch file, 122, which can be used with version A of the software application to produce version B. In this example, version A would be the "source" state and version B would be the "destination" state of the application. This binary patch file 122 can either be an executable file which acts directly on version A of the software application, or it can be a data file which is used by a separate binary patch program (not shown) to transform version A of the software application to version B." and Waldin col 6, line 48 – col 7, line 25, "In the case of virus

definition updates, there are often updates which are not operating system-specific, and sometimes there are updates which are not even computer architecture-specific. Other times, updates are specific to these, and other, categories. A single update DeltaPackage 122 may be useful to update some flavors of an application, but not others. To handle these complexities, update catalogs, referred to as "DeltaCatalogs," are utilized. These update catalogs are another example of what are referred to herein as "update patches." Rather than having a single DeltaPackage 122 correspond to each incremental update (i.e. ".DELTA.IS") as above, a DeltaCatalog corresponds to each incremental update (i.e. ".DELTA.IS"). Each DeltaCatalog has an associated source state and an associated destination state, and specifies the necessary update information by specifying which DeltaPackages 122 should be used by each flavor of the application to update from the source state to the destination state. In one embodiment, DeltaPackages 122 are given unique IDs which do not conform to the ".DELTA.AB" format used above for illustrative purposes, and are specified by the DeltaCatalogs using these unique IDs. With DeltaCatalogs substituted for DeltaPackages 122, the general scheme described above is utilized. There are a

number of different ways DeltaCatalogs can be implemented. In this embodiment, the Extensible Markup Language (XML) standard is used to create a document type definition. The XML standard is available from W3C Publications, World Wide Web Consortium, Massachusetts Institute of Technology, Laboratory for Computing Sciences, NE43-356, 545 Technology Square, Cambridge, Mass. 02139. An example document type definition corresponding to the XML standard, referred to as DPML (for DeltaPackage Markup Language), is given in Appendix A. In this document type definition, there are a number of types of entries a DeltaCatalog may contain. These types are Product (the type of software application), Package (a specific DeltaPackage 122), OS (operating system), CPU (computer architecture) and Language (the language spoken by the users of the software application). An entry of any of these types except Package may in turn contain entries of the types Product, Package, OS, CPU or Language. None of the entry types may contain a DeltaCatalog, and the Package must contain an "ID" which corresponds to a specific DeltaPackage 122. Also, the "to", or destination state, data field and the "from", or source state, data field must be given for a DeltaCatalog." Note the

variation point would be the .DELTA.IS that can update a software from version A to B)

- subsequently converting only at least one variation point of the hybrid form of the original software as necessary by a transformation in accordance with transformation rules into at least one other code formulated in the meta markup language; (Waldin col 5, line 26 – 35, "FIG. 3 illustrates this procedure for a case in which it is desired to transform an application of state F to an application of state T. Following the procedure described above, such a transformation is accomplished through only four incremental updates, from F to G to J to S to T. Each time a new DeltaPackage 122 is to be selected, the one chosen is the highest tier DeltaPackage 122 with the current application state as a source state, and a destination state which does not exceed the desired ending state.")

- and forming modified variation point of an adapted piece of at least one of software and a source code from said other code via a converter and then forming at least one of a binary and byte code of the modified variation point of an adapted piece of software via a compiler, the original and the adapted software differing in terms of at least one of their program execution and program content. (Waldin col 5, line 36 – 65, "When fewer incremental updates are required to perform a given transformation, fewer

DeltaPackages 122, and therefore less information, needs to be transferred to the application. The procedure described above produces a desired transformation using the smallest number of available DeltaPackages 122, as long as one condition is met: no available DeltaPackage 122 may have a source state which is between the source and destination states of an available DeltaPackage 122 with a lower tier. As long as this condition is met, then the procedure described above will perform an optimum transformation, using the smallest number of available DeltaPackages 122 to get from the beginning state to the desired ending state. If the condition is not met then the procedure described above may result in a transformation which uses more of the available DeltaPackages 122 than necessary. An example of a sub-optimal transformation is illustrated in FIG. 4. In that case, a transformation from state G to state S uses four DeltaPackages 122 (.DELTA.GJ, .DELTA.JM, .DELTA.MP and .DELTA.PS), when it need only use three (.DELTA.GH, .DELTA.HI, and .DELTA.IS). Because the .DELTA.IS DeltaPackage 122 has a source state (I) which is between the source and destination states of a lower tier DeltaPackage (.DELTA.GJ), the .DELTA.IS DeltaPackage 122 violates the above condition, and a sub-optimal set of

DeltaPackages 122 is used. In practice, a software publisher may easily ensure that the available DeltaPackages 122 meet this condition, since each DeltaPackage 122 is produced later in time than DeltaPackages 122 with earlier destination states. In the above example, before issuing DeltaPackage .DELTA.IS, the publisher would eliminate DeltaPackage .DELTA.GJ and possibly replace it with another, such as DeltaPackage .DELTA.GI.")

As per claim 2, Waldin discloses:

- The method as claimed in claim 1, wherein the transformation rules have at least one modification rule for a variation point. (Waldin col 5, line 26 – 65, "FIG. 3 illustrates this procedure for a case in which it is desired to transform an application of state F to an application of state T. Following the procedure described above, such a transformation is accomplished through only four incremental updates, from F to G to J to S to T. Each time a new DeltaPackage 122 is to be selected, the one chosen is the highest tier DeltaPackage 122 with the current application state as a source state, and a destination state which does not exceed the desired ending state. When fewer incremental updates are required to

perform a given transformation, fewer DeltaPackages 122, and therefore less information, needs to be transferred to the application. The procedure described above produces a desired transformation using the smallest number of available DeltaPackages 122, as long as one condition is met: no available DeltaPackage 122 may have a source state which is between the source and destination states of an available DeltaPackage 122 with a lower tier. As long as this condition is met, then the procedure described above will perform an optimum transformation, using the smallest number of available DeltaPackages 122 to get from the beginning state to the desired ending state. If the condition is not met then the procedure described above may result in a transformation which uses more of the available DeltaPackages 122 than necessary. An example of a sub-optimal transformation is illustrated in FIG. 4. In that case, a transformation from state G to state S uses four DeltaPackages 122 (.DELTA.GJ, .DELTA.JM, .DELTA.MP and .DELTA.PS), when it need only use three (.DELTA.GH, .DELTA.HI, and .DELTA.IS). Because the .DELTA.IS DeltaPackage 122 has a source state (I) which is between the source and destination states of a lower tier DeltaPackage (.DELTA.GJ), the .DELTA.IS DeltaPackage 122

violates the above condition, and a sub-optimal set of DeltaPackages 122 is used. In practice, a software publisher may easily ensure that the available DeltaPackages 122 meet this condition, since each DeltaPackage 122 is produced later in time than DeltaPackages 122 with earlier destination states. In the above example, before issuing DeltaPackage .DELTA.IS, the publisher would eliminate DeltaPackage .DELTA.GJ and possibly replace it with another, such as DeltaPackage .DELTA.GI.")

As per claim 3, Waldin discloses:

- The method as claimed in claim 1, wherein the modification rule initiates an update to at least one of a more recent software version or a patching operation. (Waldin col 4, line 17 – 35, "Each time an updated software application 110 is produced by the virus protection software publisher, the updated form of the software application constitutes a new version. The software publisher uses an incremental update builder, such as binary patch file builder 120, to produce at least one incremental update, such as binary patch file 122, which can transform a previous version of the software application to the current version. A binary patch file

builder 120 is a program which takes two versions of a software application, for example versions A and B, and produces a binary patch file, 122, which can be used with version A of the software application to produce version B. In this example, version A would be the "source" state and version B would be the "destination" state of the application. This binary patch file 122 can either be an executable file which acts directly on version A of the software application, or it can be a data file which is used by a separate binary patch program (not shown) to transform version A of the software application to version B.")

As per claim 4, Waldin discloses:

- The method as claimed in claim 1, wherein the modification of at least one variation point is performed by way of the transformation at runtime. (Waldin col 4, line 17 – 35, "Each time an updated software application 110 is produced by the virus protection software publisher, the updated form of the software application constitutes a new version. The software publisher uses an incremental update builder, such as binary patch file builder 120, to produce at least one incremental update, such as binary patch file 122, which can transform a previous version of the software

application to the current version. A binary patch file builder 120 is a program which takes two versions of a software application, for example versions A and B, and produces a binary patch file, 122, which can be used with version A of the software application to produce version B. In this example, version A would be the "source" state and version B would be the "destination" state of the application. This binary patch file 122 can either be an executable file which acts directly on version A of the software application, or it can be a data file which is used by a separate binary patch program (not shown) to transform version A of the software application to version B.")

As per claim 22, claim 22 is the non-transitory computer readable medium version of claim 1 and is therefore rejected under the same rationale.

As per claim 26, claim 26 is the non-transitory computer readable medium version of claim 3 and is therefore rejected under the same rationale.

As per claim 27, claim 27 is the non-transitory computer readable medium version of claim 4 and is therefore rejected under the same rationale.

9. Claims **6, 7 and 23** are rejected under 35 U.S.C. 102(b) as being anticpated over **Germon, "Using XML as an Intermediate Form for Compiler Development", XML conference and Exposition, 2001**.

As per claim 6, Germon discloses: A method for modifying source code, comprising:

- making a first code formulated in a meta markup language with language extensions formulated in at least one meta markup language available as the source code; (Germon page 5, second last paragraph, "`In the implementation, template-based programming can be a useful technique for translating out of XML into other formats. Each node in the tree is associated with a template with placeholders. Processes which act on the content of the node act essentially as data generators which produce fill-in values for the template.`" Note the place holder is the language extension)

- converting the source code, via a transformation in accordance with transformation rules into a second code formulated in the meta markup language without the language extensions formulated in the meta markup language; (Germon page 5, last paragraph, lines 7 – 9, "`Another possible solution is generating a template file in one pass, and resolving the placeholders explicitly in a second pass.`" Note filling in the placeholders would get rid of the language extensions)

- using the transformation rules to form a language converter which at least one of resolves and applies the language extensions of the first code in such a way that they can be processed by a back-converter that has no corresponding language extension; (Germon page 1, second paragraph, "The paper describes a small programming language prototype project, and the background theory that underlies the implementation techniques. The project involves compiling a simple block-structure imperative programming language similar to Pascal or C. XML is used as the encoding format between the parser and the code-generator. The parser compiler phase is implemented as a stand alone OmniMark program which parses the source language and outputs an XML document representing the program. The code generation phase (or backend) is a second OmniMark program which translates the XML encoded parse tree into an abstract machine language similar to assembler language for real machines." Note that machine language has no language extensions)

- and converting said second code into a second source code that is formulated in at least one of the first programming language and a different programming language and yields at least one of a valid binary code and byte code; (Germon page 1, second paragraph, "The paper describes a small programming language prototype project, and the background theory that

underlies the implementation techniques. The project involves compiling a simple block-structure imperative programming language similar to Pascal or C. XML is used as the encoding format between the parser and the code-generator. The parser compiler phase is implemented as a stand alone OmniMark program which parses the source language and outputs an XML document representing the program. The code generation phase (or backend) is a second OmniMark program which translates the XML encoded parse tree into an abstract machine language similar to assembler language for real machines.")

As per claim 7, Germon discloses:

- The method as claimed in claim 6, wherein at least one language extension is at least one of newly generated in the second code and taken over from the first code and this at least one of generation and takeover is performed by the language converter. (Germon page 1, second paragraph, "The paper describes a small programming language prototype project, and the background theory that underlies the implementation techniques. The project involves compiling a simple block-structure imperative programming language similar to Pascal

or C. XML is used as the encoding format between the parser and the code-generator. The parser compiler phase is implemented as a stand alone OmniMark program which parses the source language and outputs an XML document representing the program. The code generation phase (or backend) is a second OmniMark program which translates the XML encoded parse tree into an abstract machine language similar to assembler language for real machines.")

As per claim 23, claim 23 is the non-transitory computer readable medium version of claim 6 and is therefore rejected under the same rationale.

Claim Rejections - 35 USC § 103

10. The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

11. Claim **5** is rejected under 35 U.S.C. 103(a) as being unpatentable over **Waldin**, in view of **Tan et al (USPAT 7536686, hereinafter Tan)**.

As per claim 5, Waldin discloses:

- The method as claimed in claim 1, wherein the programming language of the source code is Java and the meta markup language of the variation points is XML and wherein the transformation and the rule description are implemented via [XML]. (Waldin col 4, line 40 – 52, "In the illustrative embodiment, the incremental update files are binary patch files which are digitally signed compressed executable modules, and the Java ARchive (JAR) platform-independent file format, available from Sun Microsystems, is used for this purpose. Because they are digitally signed, the authenticity of the updates can be ensured. When executed, the incremental update file automatically transforms a software application from a source state to a destination state. These self-contained executable incremental update files conforming to the JAR format are referred to as "DeltaPackages" 122, and are one example of what is referred to herein as an "update patch"." And col 7, line 3 – 7, "There are a number of different ways DeltaCatalogs can be implemented. In this embodiment, the Extensible Markup Language (XML) standard is used to create a document type definition.")

Waldin did not disclose:

- Wherein the XML is implemented via XSLT and XSL.

However, Tan teaches:

- Wherein the XML is implemented via XSLT and XSL. (Tan col 41, line 48 – 65, "In line 1, in accordance with the XML standards, the document in the file is identified as an XML stylesheet (XSLT) that uses terms defined in the standard XSL translator (XSLT) file indicated by the URL of the "w3" organization at directory " /XSL/Transform/1.0" Terms defined in that file constitute an XML namespace (xmlns) that is referenced within the current document by the prefix "xsl:". Thus the <template> tag defined in the standard XSLT file, and described below, is referenced within the current document as "<xsl:template>". Note that the XSLT tag "<template>" is not the same as the web site template described above with respect to Table 3. In the following discussion, the two templates are distinguished: the XSLT template tag appears below and in Table 4 as "<xsl:template>"; and the web site template of components appears as "site-template."")

It would have been obvious for one of ordinary skill in the art at the time of invention to combine teaching of Tan into that of Waldin in order to have the transformation rule implemented via XLS and XSLT. Tan provided a motivation to support this combination (Tan col 12, line 14 – 23, "The same database device management software

installed on the IDSP platform 332, and the software on the database devices connected to it, can be used by many service providers. As with any software, the database device management software for the IDSP system will undergo incremental upgrades and replacements, denoted by different version numbers. FIG. 3B depicts a system for using the Internet to provide database software updates and other database services to one or more service providers." and col 20, line 54 – 65, "The web site building wizard appliance stores a large number of such components so that a novice user does not have to reinvent them. The components can be represented in any manner known in the art. In one embodiment, each component is described by an extensible markup language (XML) document. XML uses user-definable tags to mark sections of text and other resources. Resources that are not text may be identified within an XML document by text that designates their address, such as their URL address. The XML tags are used to define one or more database objects, such as tables, queries, procedures, or indexes, that together comprise a web site database component.")

12. Claims **8 – 10, 12 – 17, 19 – 21, 24, 25, 28 and 29** are rejected under 35 U.S.C. 103(a) as being unpatentable over **Waldin**, in view of **Germon**.

As per claim 8, Waldin discloses: A method for modifying source code, comprising:

- converting a source code formulated in a first programming language into a first code formulated in a meta markup language; (Waldin col 4, line 17 – 35, "Each time an updated software application 110 is produced by the virus protection software publisher, the updated form of the software application constitutes a new version. The software publisher uses an incremental update builder, such as binary patch file builder 120, to produce at least one incremental update, such as binary patch file 122, which can transform a previous version of the software application to the current version. A binary patch file builder 120 is a program which takes two versions of a software application, for example versions A and B, and produces a binary patch file, 122, which can be used with version A of the software application to produce version B. In this example, version A would be the "source" state and version B would be the "destination" state of the application. This binary patch file 122 can either be an executable file which acts directly on version A of the software application, or it can be a data file which is used by a separate binary patch program (not shown) to transform version A of the software application to version

B." and Waldin col 6, line 48 – col 7, line 25, "In the case of virus definition updates, there are often updates which are not operating system-specific, and sometimes there are updates which are not even computer architecture-specific. Other times, updates are specific to these, and other, categories. A single update DeltaPackage 122 may be useful to update some flavors of an application, but not others. To handle these complexities, update catalogs, referred to as "DeltaCatalogs," are utilized. These update catalogs are another example of what are referred to herein as "update patches." Rather than having a single DeltaPackage 122 correspond to each incremental update (i.e. ".DELTA.IS") as above, a DeltaCatalog corresponds to each incremental update (i.e. ".DELTA.IS"). Each DeltaCatalog has an associated source state and an associated destination state, and specifies the necessary update information by specifying which DeltaPackages 122 should be used by each flavor of the application to update from the source state to the destination state. In one embodiment, DeltaPackages 122 are given unique IDs which do not conform to the ".DELTA.AB" format used above for illustrative purposes, and are specified by the DeltaCatalogs using these unique IDs. With DeltaCatalogs substituted for DeltaPackages 122,

the general scheme described above is utilized. There are a number of different ways DeltaCatalogs can be implemented. In this embodiment, the Extensible Markup Language (XML) standard is used to create a document type definition. The XML standard is available from W3C Publications, World Wide Web Consortium, Massachusetts Institute of Technology, Laboratory for Computing Sciences, NE43-356, 545 Technology Square, Cambridge, Mass. 02139. An example document type definition corresponding to the XML standard, referred to as DPML (for DeltaPackage Markup Language), is given in Appendix A. In this document type definition, there are a number of types of entries a DeltaCatalog may contain. These types are Product (the type of software application), Package (a specific DeltaPackage 122), OS (operating system), CPU (computer architecture) and Language (the language spoken by the users of the software application). An entry of any of these types except Package may in turn contain entries of the types Product, Package, OS, CPU or Language. None of the entry types may contain a DeltaCatalog, and the Package must contain an "ID" which corresponds to a specific DeltaPackage 122. Also, the "to", or destination state, data field and the "from", or source state, data field must be given for a DeltaCatalog.")

- and transforming the [intermediate] code into a second source code formulated in at least one of the first programming language and a different programming language, the first and the second source code differing in terms of their functionality. (Waldin col 5, line 26 – 65, "FIG. 3 illustrates this procedure for a case in which it is desired to transform an application of state F to an application of state T. Following the procedure described above, such a transformation is accomplished through only four incremental updates, from F to G to J to S to T. Each time a new DeltaPackage 122 is to be selected, the one chosen is the highest tier DeltaPackage 122 with the current application state as a source state, and a destination state which does not exceed the desired ending state. When fewer incremental updates are required to perform a given transformation, fewer DeltaPackages 122, and therefore less information, needs to be transferred to the application. The procedure described above produces a desired transformation using the smallest number of available DeltaPackages 122, as long as one condition is met: no available DeltaPackage 122 may have a source state which is between the source and destination states of an available DeltaPackage 122 with a lower tier. As long as this condition is met, then the procedure described above will

perform an optimum transformation, using the smallest number of available DeltaPackages 122 to get from the beginning state to the desired ending state. If the condition is not met then the procedure described above may result in a transformation which uses more of the available DeltaPackages 122 than necessary. An example of a sub-optimal transformation is illustrated in FIG. 4. In that case, a transformation from state G to state S uses four DeltaPackages 122 (.DELTA.GJ, .DELTA.JM, .DELTA.MP and .DELTA.PS), when it need only use three (.DELTA.GH, .DELTA.HI, and .DELTA.IS). Because the .DELTA.IS DeltaPackage 122 has a source state (I) which is between the source and destination states of a lower tier DeltaPackage (.DELTA.GJ), the .DELTA.IS DeltaPackage 122 violates the above condition, and a sub-optimal set of DeltaPackages 122 is used. In practice, a software publisher may easily ensure that the available DeltaPackages 122 meet this condition, since each DeltaPackage 122 is produced later in time than DeltaPackages 122 with earlier destination states. In the above example, before issuing DeltaPackage .DELTA.IS, the publisher would eliminate DeltaPackage .DELTA.GJ and possibly replace it with another, such as DeltaPackage

.DELTA.GI." and col 4, line 1 – 12, "Referring to FIG. 1, a virus protection software application 110 which incorporates a number of virus detecting routines 112, and utilizes a number of data files containing virus information 114, is installed on a user's computer 116. Because of the rate at which new viruses are created, it is desirable to update the virus protection software applications on the user's computer frequently. These updates could take place as often as daily, or even more frequently if desired. Generally, these updated applications 110 will include only small changes to the data files 114, but sometimes larger changes to the virus detecting routines 112 will also be included.")

Waldin did not disclose:

- wherein the intermediate code is obtained from transforming the first code exclusively in accordance with transformation rules, into a second code formulated in the meta markup language;

However, Germon teaches:

- wherein the intermediate code is obtained from transforming the first code exclusively in accordance with transformation rules, into a second code formulated in the meta markup language; (Germon page 5, second last

paragraph, "In the implementation, template-based programming can be a useful technique for translating out of XML into other formats. Each node in the tree is associated with a template with placeholders. Processes which act on the content of the node act essentially as data generators which produce fill-in values for the template." And last paragraph, lines 7 – 9, "Another possible solution is generating a template file in one pass, and resolving the placeholders explicitly in a second pass." Note that since the second pass is used to fill in the placeholders generated from the first pass, the second pass would result in code that is in the same language as the code generated from the first pass)

It would have been obvious for one of ordinary skill in the art at the time of invention to combine teaching of Germon into that of Waldin in order to have the intermediate code is obtained from transforming the first code exclusively in accordance with transformation rules, into a second code formulated in the meta markup language. Germon provided a motivation to support this combination (Germon page 5, second last paragraph, "In the implementation, template-based programming can be a useful technique for translating out of XML into other formats. Each node in the tree is associated with a template with placeholders. Processes which act on the content of the node act essentially as data generators which produce fill-in

values for the template." And last paragraph, lines 7 – 9, "Another possible solution is generating a template file in one pass, and resolving the placeholders explicitly in a second pass.")

As per claim 9, Waldin and Germon teach:

- The method as claimed in claim 8, wherein the transformation rules include at least one condition and at least one of one logic component and code fragment itself. (Waldin col 5, line 26 – 65, "FIG. 3 illustrates this procedure for a case in which it is desired to transform an application of state F to an application of state T. Following the procedure described above, such a transformation is accomplished through only four incremental updates, from F to G to J to S to T. Each time a new DeltaPackage 122 is to be selected, the one chosen is the highest tier DeltaPackage 122 with the current application state as a source state, and a destination state which does not exceed the desired ending state. When fewer incremental updates are required to perform a given transformation, fewer DeltaPackages 122, and therefore less information, needs to be transferred to the application. The procedure described above produces a desired

transformation using the smallest number of available DeltaPackages 122, as long as one condition is met: no available DeltaPackage 122 may have a source state which is between the source and destination states of an available DeltaPackage 122 with a lower tier. As long as this condition is met, then the procedure described above will perform an optimum transformation, using the smallest number of available DeltaPackages 122 to get from the beginning state to the desired ending state. If the condition is not met then the procedure described above may result in a transformation which uses more of the available DeltaPackages 122 than necessary. An example of a sub-optimal transformation is illustrated in FIG. 4. In that case, a transformation from state G to state S uses four DeltaPackages 122 (.DELTA.GJ, .DELTA.JM, .DELTA.MP and .DELTA.PS), when it need only use three (.DELTA.GH, .DELTA.HI, and .DELTA.IS). Because the .DELTA.IS DeltaPackage 122 has a source state (I) which is between the source and destination states of a lower tier DeltaPackage (.DELTA.GJ), the .DELTA.IS DeltaPackage 122 violates the above condition, and a sub-optimal set of DeltaPackages 122 is used. In practice, a software publisher may easily ensure that the available

DeltaPackages 122 meet this condition, since each DeltaPackage 122 is produced later in time than DeltaPackages 122 with earlier destination states. In the above example, before issuing DeltaPackage .DELTA.IS, the publisher would eliminate DeltaPackage .DELTA.GJ and possibly replace it with another, such as DeltaPackage .DELTA.GI.")

As per claim 10, Waldin and Germon teach:

- The method as claimed in claim 8, wherein transformation rules include at least one fragment in the form of at least one of a template and at least one pattern in which at least one code modification is effected with the aid of the transformation. (Germon page 5, second last paragraph, "In the implementation, template-based programming can be a useful technique for translating out of XML into other formats. Each node in the tree is associated with a template with placeholders. Processes which act on the content of the node act essentially as data generators which produce fill-in values for the template." And last paragraph, lines 7 – 9, "Another possible solution is generating a template file in one pass, and resolving the placeholders explicitly in a second pass.")

As per claim 12, Waldin and Germon teach:

- The method as claimed in claim 8, wherein at least one template is formed from at least one of the first code and a fragment of the first code with the aid of the transformation. (Germon page 5, second last paragraph, "In the implementation, template-based programming can be a useful technique for translating out of XML into other formats. Each node in the tree is associated with a template with placeholders. Processes which act on the content of the node act essentially as data generators which produce fill-in values for the template." And last paragraph, lines 7 – 9, "Another possible solution is generating a template file in one pass, and resolving the placeholders explicitly in a second pass.")

As per claim 13, Waldin discloses: A method for modifying source code, comprising:

- adding an item of information formulated in the meta markup language and influencing the subsequent program execution, via a transformation, the first code in at least one of a substituting and non-substituting way and wherein in this way, a second code also formulated in the meta markup language is formed, the transformation being performed in accordance with transformation rules formulated in a transformation description language; (Waldin col 4, line 17 – 35,

"Each time an updated software application 110 is produced by the virus protection software publisher, the updated form of the software application constitutes a new version. The software publisher uses an incremental update builder, such as binary patch file builder 120, to produce at least one incremental update, such as binary patch file 122, which can transform a previous version of the software application to the current version. A binary patch file builder 120 is a program which takes two versions of a software application, for example versions A and B, and produces a binary patch file, 122, which can be used with version A of the software application to produce version B. In this example, version A would be the "source" state and version B would be the "destination" state of the application. This binary patch file 122 can either be an executable file which acts directly on version A of the software application, or it can be a data file which is used by a separate binary patch program (not shown) to transform version A of the software application to version B." and Waldin col 6, line 48 – col 7, line 25, "In the case of virus definition updates, there are often updates which are not operating system-specific, and sometimes there are updates which are not even computer architecture-specific. Other

times, updates are specific to these, and other, categories. A single update DeltaPackage 122 may be useful to update some flavors of an application, but not others. To handle these complexities, update catalogs, referred to as "DeltaCatalogs," are utilized. These update catalogs are another example of what are referred to herein as "update patches." Rather than having a single DeltaPackage 122 correspond to each incremental update (i.e. ".DELTA.IS") as above, a DeltaCatalog corresponds to each incremental update (i.e. ".DELTA.IS"). Each DeltaCatalog has an associated source state and an associated destination state, and specifies the necessary update information by specifying which DeltaPackages 122 should be used by each flavor of the application to update from the source state to the destination state. In one embodiment, DeltaPackages 122 are given unique IDs which do not conform to the ".DELTA.AB" format used above for illustrative purposes, and are specified by the DeltaCatalogs using these unique IDs. With DeltaCatalogs substituted for DeltaPackages 122, the general scheme described above is utilized. There are a number of different ways DeltaCatalogs can be implemented. In this embodiment, the Extensible Markup Language (XML) standard is used to create a document type definition. The

XML standard is available from W3C Publications, World Wide Web Consortium, Massachusetts Institute of Technology, Laboratory for Computing Sciences, NE43-356, 545 Technology Square, Cambridge, Mass. 02139. An example document type definition corresponding to the XML standard, referred to as DPML (for DeltaPackage Markup Language), is given in Appendix A. In this document type definition, there are a number of types of entries a DeltaCatalog may contain. These types are Product (the type of software application), Package (a specific DeltaPackage 122), OS (operating system), CPU (computer architecture) and Language (the language spoken by the users of the software application). An entry of any of these types except Package may in turn contain entries of the types Product, Package, OS, CPU or Language. None of the entry types may contain a DeltaCatalog, and the Package must contain an "ID" which corresponds to a specific DeltaPackage 122. Also, the "to", or destination state, data field and the "from", or source state, data field must be given for a DeltaCatalog.")

- and transforming the second code into a second source code formulated in at least one of the first programming language and a different programming language, at least one of the program content and program execution of the first

source code differing from at least one of the program content and program execution of the second source code. (Waldin col 5, line 26 – 65, "FIG. 3 illustrates this procedure for a case in which it is desired to transform an application of state F to an application of state T. Following the procedure described above, such a transformation is accomplished through only four incremental updates, from F to G to J to S to T. Each time a new DeltaPackage 122 is to be selected, the one chosen is the highest tier DeltaPackage 122 with the current application state as a source state, and a destination state which does not exceed the desired ending state. When fewer incremental updates are required to perform a given transformation, fewer DeltaPackages 122, and therefore less information, needs to be transferred to the application. The procedure described above produces a desired transformation using the smallest number of available DeltaPackages 122, as long as one condition is met: no available DeltaPackage 122 may have a source state which is between the source and destination states of an available DeltaPackage 122 with a lower tier. As long as this condition is met, then the procedure described above will perform an optimum transformation, using the smallest number of available DeltaPackages 122 to get from the

beginning state to the desired ending state. If the condition is not met then the procedure described above may result in a transformation which uses more of the available DeltaPackages 122 than necessary. An example of a sub-optimal transformation is illustrated in FIG. 4. In that case, a transformation from state G to state S uses four DeltaPackages 122 (.DELTA.GJ, .DELTA.JM, .DELTA.MP and .DELTA.PS), when it need only use three (.DELTA.GH, .DELTA.HI, and .DELTA.IS). Because the .DELTA.IS DeltaPackage 122 has a source state (I) which is between the source and destination states of a lower tier DeltaPackage (.DELTA.GJ), the .DELTA.IS DeltaPackage 122 violates the above condition, and a sub-optimal set of DeltaPackages 122 is used. In practice, a software publisher may easily ensure that the available DeltaPackages 122 meet this condition, since each DeltaPackage 122 is produced later in time than DeltaPackages 122 with earlier destination states. In the above example, before issuing DeltaPackage .DELTA.IS, the publisher would eliminate DeltaPackage .DELTA.GJ and possibly replace it with another, such as DeltaPackage .DELTA.GI." and col 4, line 1 – 12, "Referring to FIG. 1, a virus protection software application 110 which incorporates a

number of virus detecting routines 112, and utilizes a number of data files containing virus information 114, is installed on a user's computer 116. Because of the rate at which new viruses are created, it is desirable to update the virus protection software applications on the user's computer frequently. These updates could take place as often as daily, or even more frequently if desired. Generally, these updated applications 110 will include only small changes to the data files 114, but sometimes larger changes to the virus detecting routines 112 will also be included.")

Waldin did not disclose:

- converting a source code, formulated in a first programming language into a first code formulated in a meta markup language;

However, Germon teaches:

- converting a source code, formulated in a first programming language into a first code formulated in a meta markup language; (Germon page 1, second paragraph, "The paper describes a small programming language prototype project, and the background theory that underlies the implementation techniques. The project involves compiling a simple block-structure imperative programming

language similar to Pascal or C. XML is used as the encoding format between the parser and the code-generator. The parser compiler phase is implemented as a stand alone OmniMark program which parses the source language and outputs an XML document representing the program. The code generation phase (or backend) is a second OmniMark program which translates the XML encoded parse tree into an abstract machine language similar to assembler language for real machines.")

It would have been obvious for one of ordinary skill in the art at the time of invention to combine teaching of Germon into that of Waldin in order to have converting a source code, formulated in a first programming language into a first code formulated in a meta markup language. Germon provided a motivation to support this combination (Germon page 1, third paragraph, "Although strictly speaking, the paper is about compiler implementation, the techniques are far more broadly useful. The first pass (parser) is all about overlaying a structure on a serial encoding of data, conversion of non-XML to XML requires the same techniques." And fourth paragraph, "XML was used as the intermediate form fro this project for a number of reasons. First, since XML is a text based encoding, the tree encoding would be human-readable. This would allow experimenting with optimization techniques, and one would be able to read the

resulting trees at all stages in the process. Second, I was interested in exploring parallels between the traditional data conversion problem domain and the programming language implementation problem domain. Finally, XML is designed to encode trees, and XML tools are designed to manipulate those trees. In other words, XML and XML tools are well suited to implementing the back end portion of a programming compiler.")

As per claim 14, Waldin and Germon teach:

- The method as claimed in claim 13, wherein said information includes at least one code fragment and wherein the second source code is formed in that at least one code fragment contained in the first source code is replaced with the aid of the transformation by the at least one code fragment contained in the fragment. (Waldin col 6, line 48 – col 7, line 25, "In the case of virus definition updates, there are often updates which are not operating system-specific, and sometimes there are updates which are not even computer architecture-specific. Other times, updates are specific to these, and other, categories. A single update DeltaPackage 122 may be useful to update some flavors of an application, but not others. To handle these complexities, update catalogs, referred to as "DeltaCatalogs," are utilized. These update catalogs are

another example of what are referred to herein as "update patches." Rather than having a single DeltaPackage 122 correspond to each incremental update (i.e. ".DELTA.IS") as above, a DeltaCatalog corresponds to each incremental update (i.e. ".DELTA.IS"). Each DeltaCatalog has an associated source state and an associated destination state, and specifies the necessary update information by specifying which DeltaPackages 122 should be used by each flavor of the application to update from the source state to the destination state. In one embodiment, DeltaPackages 122 are given unique IDs which do not conform to the ".DELTA.AB" format used above for illustrative purposes, and are specified by the DeltaCatalogs using these unique IDs. With DeltaCatalogs substituted for DeltaPackages 122, the general scheme described above is utilized. There are a number of different ways DeltaCatalogs can be implemented. In this embodiment, the Extensible Markup Language (XML) standard is used to create a document type definition. The XML standard is available from W3C Publications, World Wide Web Consortium, Massachusetts Institute of Technology, Laboratory for Computing Sciences, NE43-356, 545 Technology Square, Cambridge, Mass. 02139. An example document type definition corresponding to the XML standard, referred to

as DPML (for DeltaPackage Markup Language), is given in Appendix A. In this document type definition, there are a number of types of entries a DeltaCatalog may contain. These types are Product (the type of software application), Package (a specific DeltaPackage 122), OS (operating system), CPU (computer architecture) and Language (the language spoken by the users of the software application). An entry of any of these types except Package may in turn contain entries of the types Product, Package, OS, CPU or Language. None of the entry types may contain a DeltaCatalog, and the Package must contain an "ID" which corresponds to a specific DeltaPackage 122. Also, the "to", or destination state, data field and the "from", or source state, data field must be given for a DeltaCatalog.")

As per claim 15, Waldin and Germon teach:

- The method as claimed in claim 13, wherein the information specifically includes data in the form of at least one of initialization states, state data and database data. (Waldin col 5, line 26 – 65, "FIG. 3 illustrates this procedure for a case in which it is desired to transform an application of state F to an application of state T. Following the procedure described above, such a

transformation is accomplished through only four incremental updates, from F to G to J to S to T. Each time a new DeltaPackage 122 is to be selected, the one chosen is the highest tier DeltaPackage 122 with the current application state as a source state, and a destination state which does not exceed the desired ending state. When fewer incremental updates are required to perform a given transformation, fewer DeltaPackages 122, and therefore less information, needs to be transferred to the application. The procedure described above produces a desired transformation using the smallest number of available DeltaPackages 122, as long as one condition is met: no available DeltaPackage 122 may have a source state which is between the source and destination states of an available DeltaPackage 122 with a lower tier. As long as this condition is met, then the procedure described above will perform an optimum transformation, using the smallest number of available DeltaPackages 122 to get from the beginning state to the desired ending state. If the condition is not met then the procedure described above may result in a transformation which uses more of the available DeltaPackages 122 than necessary. An example of a sub-optimal transformation is illustrated in FIG. 4. In that

case, a transformation from state G to state S uses four DeltaPackages 122 (.DELTA.GJ, .DELTA.JM, .DELTA.MP and .DELTA.PS), when it need only use three (.DELTA.GH, .DELTA.HI, and .DELTA.IS). Because the .DELTA.IS DeltaPackage 122 has a source state (I) which is between the source and destination states of a lower tier DeltaPackage (.DELTA.GJ), the .DELTA.IS DeltaPackage 122 violates the above condition, and a sub-optimal set of DeltaPackages 122 is used. In practice, a software publisher may easily ensure that the available DeltaPackages 122 meet this condition, since each DeltaPackage 122 is produced later in time than DeltaPackages 122 with earlier destination states. In the above example, before issuing DeltaPackage .DELTA.IS, the publisher would eliminate DeltaPackage .DELTA.GJ and possibly replace it with another, such as DeltaPackage .DELTA.GI." and col 4, line 1 – 12, "Referring to FIG. 1, a virus protection software application 110 which incorporates a number of virus detecting routines 112, and utilizes a number of data files containing virus information 114, is installed on a user's computer 116. Because of the rate at which new viruses are created, it is desirable to update the virus protection software applications on the user's

computer frequently. These updates could take place as often as daily, or even more frequently if desired. Generally, these updated applications 110 will include only small changes to the data files 114, but sometimes larger changes to the virus detecting routines 112 will also be included.")

As per claim 16, Waldin and Germon teach:

- The method as claimed in claim 15, wherein the transformation rules are influenced by the data. (Waldin col 5, line 26 – 65, "FIG. 3 illustrates this procedure for a case in which it is desired to transform an application of state F to an application of state T. Following the procedure described above, such a transformation is accomplished through only four incremental updates, from F to G to J to S to T. Each time a new DeltaPackage 122 is to be selected, the one chosen is the highest tier DeltaPackage 122 with the current application state as a source state, and a destination state which does not exceed the desired ending state. When fewer incremental updates are required to perform a given transformation, fewer DeltaPackages 122, and therefore less information, needs to be transferred to the application.

The procedure described above produces a desired transformation using the smallest number of available DeltaPackages 122, as long as one condition is met: no available DeltaPackage 122 may have a source state which is between the source and destination states of an available DeltaPackage 122 with a lower tier. As long as this condition is met, then the procedure described above will perform an optimum transformation, using the smallest number of available DeltaPackages 122 to get from the beginning state to the desired ending state. If the condition is not met then the procedure described above may result in a transformation which uses more of the available DeltaPackages 122 than necessary. An example of a sub-optimal transformation is illustrated in FIG. 4. In that case, a transformation from state G to state S uses four DeltaPackages 122 (.DELTA.GJ, .DELTA.JM, .DELTA.MP and .DELTA.PS), when it need only use three (.DELTA.GH, .DELTA.HI, and .DELTA.IS). Because the .DELTA.IS DeltaPackage 122 has a source state (I) which is between the source and destination states of a lower tier DeltaPackage (.DELTA.GJ), the .DELTA.IS DeltaPackage 122 violates the above condition, and a sub-optimal set of DeltaPackages 122 is used. In practice, a software

publisher may easily ensure that the available DeltaPackages 122 meet this condition, since each DeltaPackage 122 is produced later in time than DeltaPackages 122 with earlier destination states. In the above example, before issuing DeltaPackage .DELTA.IS, the publisher would eliminate DeltaPackage .DELTA.GJ and possibly replace it with another, such as DeltaPackage .DELTA.GI." and col 4, line 1 – 12, "Referring to FIG. 1, a virus protection software application 110 which incorporates a number of virus detecting routines 112, and utilizes a number of data files containing virus information 114, is installed on a user's computer 116. Because of the rate at which new viruses are created, it is desirable to update the virus protection software applications on the user's computer frequently. These updates could take place as often as daily, or even more frequently if desired. Generally, these updated applications 110 will include only small changes to the data files 114, but sometimes larger changes to the virus detecting routines 112 will also be included.")

As per claim 17, Waldin and Germon teach:

- The method as claimed in claim 13, wherein at least one of the data and code fragments are additionally embedded in the transformation rules. (Waldin col 5, line 26 – 65, "FIG. 3 illustrates this procedure for a case in which it is desired to transform an application of state F to an application of state T. Following the procedure described above, such a transformation is accomplished through only four incremental updates, from F to G to J to S to T. Each time a new DeltaPackage 122 is to be selected, the one chosen is the highest tier DeltaPackage 122 with the current application state as a source state, and a destination state which does not exceed the desired ending state. When fewer incremental updates are required to perform a given transformation, fewer DeltaPackages 122, and therefore less information, needs to be transferred to the application. The procedure described above produces a desired transformation using the smallest number of available DeltaPackages 122, as long as one condition is met: no available DeltaPackage 122 may have a source state which is between the source and destination states of an available DeltaPackage 122 with a lower tier. As long as this condition is met, then the procedure described above will perform an optimum transformation, using the smallest number of available DeltaPackages 122 to get from the

beginning state to the desired ending state. If the condition is not met then the procedure described above may result in a transformation which uses more of the available DeltaPackages 122 than necessary. An example of a sub-optimal transformation is illustrated in FIG. 4. In that case, a transformation from state G to state S uses four DeltaPackages 122 (.DELTA.GJ, .DELTA.JM, .DELTA.MP and .DELTA.PS), when it need only use three (.DELTA.GH, .DELTA.HI, and .DELTA.IS). Because the .DELTA.IS DeltaPackage 122 has a source state (I) which is between the source and destination states of a lower tier DeltaPackage (.DELTA.GJ), the .DELTA.IS DeltaPackage 122 violates the above condition, and a sub-optimal set of DeltaPackages 122 is used. In practice, a software publisher may easily ensure that the available DeltaPackages 122 meet this condition, since each DeltaPackage 122 is produced later in time than DeltaPackages 122 with earlier destination states. In the above example, before issuing DeltaPackage .DELTA.IS, the publisher would eliminate DeltaPackage .DELTA.GJ and possibly replace it with another, such as DeltaPackage .DELTA.GI." and col 4, line 1 – 12, "Referring to FIG. 1, a virus protection software application 110 which incorporates a

number of virus detecting routines 112, and utilizes a number of data files containing virus information 114, is installed on a user's computer 116. Because of the rate at which new viruses are created, it is desirable to update the virus protection software applications on the user's computer frequently. These updates could take place as often as daily, or even more frequently if desired. Generally, these updated applications 110 will include only small changes to the data files 114, but sometimes larger changes to the virus detecting routines 112 will also be included.")

As per claim 19, Waldin and Germon teach:

- The method as claimed in claim 13, wherein information includes at least one of updates and patches. (Waldin col 4, line 1 – 12, "Referring to FIG. 1, a virus protection software application 110 which incorporates a number of virus detecting routines 112, and utilizes a number of data files containing virus information 114, is installed on a user's computer 116. Because of the rate at which new viruses are created, it is desirable to update the virus protection software applications on the user's computer frequently. These

updates could take place as often as daily, or even more frequently if desired. Generally, these updated applications 110 will include only small changes to the data files 114, but sometimes larger changes to the virus detecting routines 112 will also be included.")

As per claim 20, Waldin and Germon teach:

- The method as claimed in claim 13, wherein fragments contain internationalization information which serves for the adaptation to different natural languages. (Waldin col 6, line 37 – 47, "Software publishers often produce different "flavors" of a single software application, directed to different computer architectures, different operating systems, and users who speak different languages. The scheme for publishing incremental updates laid out above is adequate for the case in which there is only one flavor of a software application. For the more general case of several application flavors, however, some additional mechanisms can be used to handle the additional complexities of parallel updating. A system which addresses these complexities is described in the second illustrative embodiment of the present invention.")

As per claim 21, Waldin and Germon teach:

- The method as claimed in claim 13, wherein at least one of data and code fragments originate from a library and carry information tailored to at least one of customers and customer groups. (Waldin col 6, line 37 – 47, "Software publishers often produce different "flavors" of a single software application, directed to different computer architectures, different operating systems, and users who speak different languages. The scheme for publishing incremental updates laid out above is adequate for the case in which there is only one flavor of a software application. For the more general case of several application flavors, however, some additional mechanisms can be used to handle the additional complexities of parallel updating. A system which addresses these complexities is described in the second illustrative embodiment of the present invention.")

As per claim 24, claim 24 is the non-transitory computer readable medium version of claim 8 and is therefore rejected under the same rationale.

As per claim 25, claim 25 is the non-transitory computer readable medium version of claim 13 and is therefore rejected under the same rationale.

As per claim 28, claim 28 claims duplicate subject matter as claim 10 above and is therefore rejected under the same rationale.

As per claim 29, claim 29 is the non-transitory computer readable medium version of claim 12 and is therefore rejected under the same rationale.

13. Claims **11 and 18** are rejected under 35 U.S.C. 103(a) as being unpatentable over **Waldin** and **Germon**, in view of **Tan**.

As per claim 11, Waldin and Germon did not teach:

- The method as claimed in claim 10, wherein the transformation rules are embodied in such a way that a mechanism for backing up at least one system state is incorporated into the second source code with the aid of the transformation in order to enable a migration into other versions.

However, Tan teaches:

- wherein the transformation rules are embodied in such a way that a mechanism for backing up at least one system state is incorporated into the second source code with the aid of the transformation in order to enable a migration into other versions. (Tan col 55, line 20 – 51, "The backup server 1280, such as a backup catalog server, provides backup for critical information that is stored in the local network 1220. This

information about software installed on the appliances and configuration parameters is used to replace or reboot an appliance that has failed or been disconnected from the network. The management services server 1210 ensures that all management activities required to maintain a database are distributed among the database appliances available on the network. When a new appliance is added to the network, its address should be added to the directory server, information about its configuration should be added to the backup server, and the installed applications and consumable resources of the appliance should be made available to the management services server 1210. With conventional database appliances, a human operator must recognize when a new appliance 1250 has been connected to the network 1220 and must register that new appliance with the management services server 1210. The registration process includes the database appliance in the list of available appliances and makes available to the management services server the installed applications and consumable resources of the appliance. Then the human operator chooses management tasks that the new appliance should perform. The management tasks include jobs such as backing up the files, clearing out logs, monitoring space usage, creating an

index, and synchronizing content with another database, among many others known to those in the art. The management tasks also include events that trigger a response from the new appliance. For example, space usage reaching a threshold amount constitutes an event that triggers an alert being sent to a user of the database device. The management tasks also include patches to the software that was originally installed in the new appliance at the factory.")

It would have been obvious for one of ordinary skill in the art at the time of invention to combine teaching of Tan into that of Waldin and Germon in order to have the transformation rule embodied in such a way that a mechanism for backing up at least one system state is incorporated into the second source code with the aid of the transformation in order to enable a migration into other versions.. Tan provided a motivation to support this combination (Tan col 55, line 28 – 51, "When a new appliance is added to the network, its address should be added to the directory server, information about its configuration should be added to the backup server, and the installed applications and consumable resources of the appliance should be made available to the management services server 1210. With conventional database appliances, a human operator must recognize when a new appliance 1250 has been connected to the

network 1220 and must register that new appliance with the management services server 1210. The registration process includes the database appliance in the list of available appliances and makes available to the management services server the installed applications and consumable resources of the appliance. Then the human operator chooses management tasks that the new appliance should perform. The management tasks include jobs such as backing up the files, clearing out logs, monitoring space usage, creating an index, and synchronizing content with another database, among many others known to those in the art. The management tasks also include events that trigger a response from the new appliance. For example, space usage reaching a threshold amount constitutes an event that triggers an alert being sent to a user of the database device. The management tasks also include patches to the software that was originally installed in the new appliance at the factory.")

As per claim 18, Waldin and Germon did not teach:

- The method as claimed in claim 13, wherein data generated by checkpoints is added by a transformation in such a way that the internal state of the original program is at least one of available at the restart of the program and is usable by the program.

However, Tan teaches:

- wherein data generated by checkpoints is added by a transformation in such a way that the internal state of the original program is at least one of available at the restart of the program and is usable by the program. (Tan col 55, line 20 – 51, "The backup server 1280, such as a backup catalog server, provides backup for critical information that is stored in the local network 1220. This information about software installed on the appliances and configuration parameters is used to replace or reboot an appliance that has failed or been disconnected from the network. The management services server 1210 ensures that all management activities required to maintain a database are distributed among the database appliances available on the network. When a new appliance is added to the network, its address should be added to the directory server, information about its configuration should be added to the backup server, and the installed applications and consumable resources of the appliance should be made available to the management services server 1210. With conventional database appliances, a human operator must recognize when a new appliance 1250 has been connected to the network 1220 and must register that new appliance with the management services server 1210. The

registration process includes the database appliance in the list of available appliances and makes available to the management services server the installed applications and consumable resources of the appliance. Then the human operator chooses management tasks that the new appliance should perform. The management tasks include jobs such as backing up the files, clearing out logs, monitoring space usage, creating an index, and synchronizing content with another database, among many others known to those in the art. The management tasks also include events that trigger a response from the new appliance. For example, space usage reaching a threshold amount constitutes an event that triggers an alert being sent to a user of the database device. The management tasks also include patches to the software that was originally installed in the new appliance at the factory.")

It would have been obvious for one of ordinary skill in the art at the time of invention to combine teaching of Tan into that of Waldin and Germon in order to have wherein data generated by checkpoints is added by a transformation in such a way that the internal state of the original program is at least one of available at the restart of the program and is usable by the program. Tan provided a motivation to support this combination (Tan col 55, line 28 – 51, "When a new appliance is added to the network, its

address should be added to the directory server, information about its configuration should be added to the backup server, and the installed applications and consumable resources of the appliance should be made available to the management services server 1210. With conventional database appliances, a human operator must recognize when a new appliance 1250 has been connected to the network 1220 and must register that new appliance with the management services server 1210. The registration process includes the database appliance in the list of available appliances and makes available to the management services server the installed applications and consumable resources of the appliance. Then the human operator chooses management tasks that the new appliance should perform. The management tasks include jobs such as backing up the files, clearing out logs, monitoring space usage, creating an index, and synchronizing content with another database, among many others known to those in the art. The management tasks also include events that trigger a response from the new appliance. For example, space usage reaching a threshold amount constitutes an event that triggers an alert being sent to a user of the database device. The management tasks also include patches to the software that was originally installed in the new appliance at the factory.")

Response to Arguments

14. Applicant's arguments filed 1/18/2011 have been fully considered but they are not persuasive.

Applicant argued on page 12 that Waldin reference did not teach claim 1 and 22, limitation "Initially forming, from an original piece of software including only source text, a hybrid form of the original software, formed in such a way that at least one part of the source text is compiled into at least one of a byte and binary code and at least one further part of the source text is converted into a code formulated in a meta markup language for at least one variation point."

The examiner disagrees. First of all, Waldin column 4, line 17 – 52 teaches creating delta packages that allows software to be updated from version A to version B. original software application version A and B are therefore mapped to the original pieces of software including only a source text. A binary patch file is created by comparing the differences between software application version A and version B, the binary patch file can be an executable file which means its a machine executable, therefore its a binary or byte code file. Plurality of the binary patch file that applies for different systems can be batched together to form a Delta Catalog which allows updates to any type of platforms or system architecture (Waldin column 6, line 48 – column 7, line 3). The delta catalog which contains XML definition and binary patch file is therefore

mapped to the hybrid form of the original software, created from the original software to be used to update a software application from version A to version B.

Applicant argued on page 15 that Germon reference did not teach claim 6 and 23, limitations "making a first code formulated in a meta markup language with language extensions formulated in at least one meta markup language available as the source code;" and "converting the source code, via a transformation in accordance with transformation rules into a second code formulated in the meta markup language without the language extensions formulated in the meta markup language;"

The examiner disagrees. Germon page 5, last paragraph is used to reject the first limitation in question. Germon teaches that a template file can be generated at the first pass while converting XML files into other formats. In other words, the first code which is in XML is converted into a XML based templates with place holders, thus the XML based template is mapped to the source code. The template files with placeholders is then run again in a second pass with the placeholders resolved, thus leaving only the template file in XML to be able to be converted to other non-XML translation. Thus the template in XML with the placeholders resolved is mapped to the second code without the language extensions.

Applicant argued on page 17 that Germon reference did not teach claim 8 and 24, limitations "converting a source code formulated in a first programming language

into a first code formulated in a meta markup language;" and " and transforming the second code into a second source code formulated in at least one of the first programming language and a different programming language, the first and the second source code differing in terms of their functionality."

The examiner disagrees. In non final rejection dated 10/13/2010, the examiner has broken down claim 8 in 2 different parts. The first part being "converting a source code formulated in a first programming language into a first code formulated in a meta markup language; and transforming the [intermediate] code into a second source code formulated in at least one of the first programming language and a different programming language, the first and the second source code differing in terms of their functionality." the second part is "wherein the intermediate code is obtained from transforming the first code exclusively in accordance with transformation rules, into a second code formulated in the meta markup language;". Waldin column 4, line 17 – 52 teaches creating delta packages that allow software to be updated from version A to version B. original software application version A and B are therefore mapped to the original pieces of software including only a source text. A binary patch file is created by comparing the differences between software application version A and version B, the binary patch file can be an executable file which means its a machine executable, therefore its a binary or byte code file. Plurality of the binary patch file that applies for different systems can be batched together to form a Delta Catalog which allows updates to any type of platforms or system architecture (Waldin column 6, line 48

– column 7, line 3). The delta catalog which contains XML definition and binary patch file is therefore mapped to the first code, created from the original software to be used to update a software application from version A to version B. Waldin column 5, line 26 – 65 and column 4, lines 1 – 12 teaches how the delta packages and delta catalog (which contains XML definitions) are used to upgrade software from version A to version B. The examiner noted in the previous office action that Waldin did not teach the claimed second limitation "wherein the intermediate code is obtained from transforming the first code exclusively in accordance with transformation rules, into a second code formulated in the meta markup language;" that deficiency is rectified with the Germon reference where intermediate representation comprising a template file can be generated at the first pass while converting XML files into other formats. In other words, the first code which is in XML is converted into a XML based templates with place holders, thus the XML based template is mapped to the source code. The template files with placeholders is then run again in a second pass with the placeholders resolved, thus leaving only the template file in XML to be able to be converted to other non-XML translation. Thus the template in XML with the placeholders resolved is mapped to the second code or the intermediate code.

Applicant argued on page 19 that Germon reference did not teach claim 13 and 25, limitations "adding an item of information formulated in the meta markup language and influencing the subsequent program execution, via a transformation, the first code in at least one of a substituting and non-substituting way and wherein in this way, the

second code also formulated in the meta markup language is formed, the transformation being performed in accordance with transformation rules formulated in a transformation description language;" and also challenged the reason for combining Waldin and Germon reference in light of claims 13 and 25 on page 21.

The examiner disagrees. Waldin column 4, line 17 – 52 teaches creating delta packages that allow software to be updated from version A to version B. original software application version A and B are therefore mapped to the original pieces of software including only a source text. A binary patch file is created by comparing the differences between software application version A and version B. Plurality of the binary patch file that applies for different systems can be batched together to form a Delta Catalog which allows updates to any type of platforms or system architecture (Waldin column 6, line 48 – column 7, line 3). The XML definition is therefore mapped to the item of information formulated in the meta markup language and influencing the subsequent program execution. The delta catalog resulted from the combination of XML definition and binary patch file is therefore mapped to the second code used to update a software application from version A to version B.

Also Waldin reference teaches ways of updating software from version A to version B by creating binary patch files resulted from comparing the differences between version A and version B. Germon references teaches ways to converting codes into XML based format to be allowed for easier implementations. Although

Germon is primarily about compiler implementations, it can be for far broader purposes that any code can be converted into XML, thus Waldin's technique of converting the differences between version A and version B into binary patch file can also benefit from the teaching as suggested by Germon, thus enhances the overall appeal of both references.

Conclusion

15. **THIS ACTION IS MADE FINAL.** Applicant is reminded of the extension of time policy as set forth in 37 CFR 1.136(a).

A shortened statutory period for reply to this final action is set to expire THREE MONTHS from the mailing date of this action. In the event a first reply is filed within TWO MONTHS of the mailing date of this final action and the advisory action is not mailed until after the end of the THREE-MONTH shortened statutory period, then the shortened statutory period will expire on the date the advisory action is mailed, and any extension fee pursuant to 37 CFR 1.136(a) will be calculated from the mailing date of the advisory action. In no event, however, will the statutory period for reply expire later than SIX MONTHS from the mailing date of this final action.

Any inquiry concerning this communication or earlier communications from the examiner should be directed to Charles Swift whose telephone number is (571)270-7756. The examiner can normally be reached on Monday through Thursday, 9:00AM to 6:00PM, Friday 10:30AM - 3:30PM, Eastern Time.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Emerson Puente can be reached on (571)272-3652. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free). If you would like assistance from a USPTO Customer Service Representative or access to the automated information system, call 800-786-9199 (IN USA OR CANADA) or 571-272-1000.

/Charles Swift/
Examiner, Art Unit 2196

/Emerson C Puente/
Supervisory Patent Examiner, Art Unit 2196